Deploying the Robot Operating System (ROS) for Complex Path Planning on Planar Surfaces

Honors Undergraduate Research Thesis

Adam Buynak

The Ohio State University

Department of Mechanical and Aerospace Engineering

April 2021

Thesis Committee

Dr. Michael Groeber, Advisor

Mr. Walter Hansen

Dr. Samantha Krening

Abstract

Industrial robotics are characterized by intrinsically closed systems and present a unique difficulty to the factory integration efforts of the next industrial revolution. Homogenizing the controls of a factory, specifically a robotic operation, allows engineers to design for modularity. To meet this need, collaborative academic and industry efforts have developed the Robot Operating System (ROS) as a package-based programming framework. Developers assembling motion control systems have a diversity of options for key packages such as the path planner, the kinematic motion planner, the collision checker, and the robot-specific support package.

The goal of this study is to configure and deploy a motion control system for an industrial robotic cell using existing ROS packages. Further, a path planning package will be developed to navigate complex planar paths on a flat surface. Existing path planners for curved, planar-like surfaces are designed to output a raster (parallel rows of linear lines) type motion. A method for creating complex, non-linear paths across a surface is needed to support applications such as non-destructive inspection and surface finishing. To provide context to this study, a demonstrative application of navigating a robotic arm through a grid-based maze presented to the robot work cell was employed. Initial work configured the software environment including the robot support, motion planning, and sensor packages necessary to form the motion planning pipeline and testing tools.

Machine vision was enabled through the use of an RGB/stereo depth camera and static frames were filtered with the DREAM.3D data processing software. Filtered data was used to solve and navigate the presented maze. A tool for generating and visualizing these three-dimensional pathways was created. This study successfully stood up a motion control system based on the MoveIt planning framework and deployed a novel planar path planning technique. Analysis determined the MoveIt framework to be insufficient for a semi-constrained, short throw paths and suggested future work to explore alternative motion planners. With many existing tools focusing on parallel paths, this tool offers greater flexibility to system designers in achieving unique, complex motion.

Acknowledgments

I would like to thank Mr. Walter Hansen at the Center for Design and Manufacturing Excellence whose practical experience and overall mentorship was invaluable to my introduction to industrial robotics.

I would like to thank Dr. Michael Groeber for serving as my research advisor. His experience in image processing and research project management helped bring this project to a successful conclusion.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 – State of Industrial Robotics

Over the past six decades, articulating robotic arms have become increasingly prevalent on manufacturing floors, maintenance depots, and industrial workspaces. Traditional applications include repetitive-action, closed-loop type operations such as product transport and machine tending. In these applications, the traditionally monolithic control software can provide adequate performance when programmed to respond to specific sensor inputs with a finite number of predefined actions. While advances in machine vision allow programs to more easily sense and adjust to small variations in the environment, applications still rely on repeatable actions to pre-taught positions.

Robotic motion control in industry and academia is achieved through proprietary software deployed on a handheld “teach pendant” (TP) or through an external computer application. The demand for greater flexibility in control options has driven development open-source controllers which are abstracted from the proprietary controller. Software development favors the standardization of modular tools to simplify code structure and enable reusability. Each manufacturer offers hardware-specific tools for joint control or cartesian point-to-point travel. Using open-source controllers allow artificially intelligent (AI) driven systems to build motion paths from similar tools. Vision tools identify objects, find regions of interest, inform motion planning, enable environment interaction,

and monitor for collisions. When assembled together, these many tools build a system which has the environmental awareness needed to robustly plan motion.

These capabilities may be achieved using specialized open-source packages connected to a top-level process manager. This process manager could be explicitly controlled by user input or by an AI program.

1.2 – An Introduction to the Robot Operating System

"The Robot Operating System (ROS) is *the* de-facto standard for robotic software" [1]. ROS is an open-source, modular framework for programming robots supported by research and industrial institutions worldwide. Having overcome the proprietary ownership hurdle, ROS provides a common interface for controllers, vision sources, and sensors to develop a closed-loop system [2]. ROS allows rapid prototyping with the ease of scaling bench level innovations to industrially robust applications. Further barriers to research and development are avoided through allowing flexible integration of key packages such as the path planner, the kinematic motion planner, the collision checker, and the robot-specific support package. This streamlines the routine tasks of a robotics integration project. The benefits are two-fold: (1) users may focus on developing application specific tools and (2) accelerate the prototyping process. ROS will be used as the foundational robotic programming environment for this study.

1.3 – Research Goals

This study will review and assemble existing ROS packages to build a motion planning and control network equipped with machine vision capabilities. Given the lack of comprehensive examples or tutorials in assembling an industrial motion network, this study will develop documentation which centralizes much of the information necessary to deploy this capability. Exiting path planning tools will be evaluated to determine capability and modularity of existing techniques. With this information, a demonstrative example for path planning will be developed as a sample use case.

1.4 – Motivation

In the ROS Industrial field, a push to develop more robust motion planning pipelines has persisted since the inception thereof. Given the nature of rapidly developing research and development in an industrial context, successfully integrated systems are generally undocumented. A study of the practices of roboticists published in May 2020 found that 16.4% of ROS projects had minimally documented their system architecture [1]. For new ROS-Industrial users, building this system architecture is a crucial first step with a steep learning curve. This underscores the need for extensive development to achieving a functional motion network.

With the sponsoring Artificially Intelligent Manufacturing Systems (AIMS) Lab launching a new field of work into open-source control, this study will form a fundamental knowledge base and capability. Future research will build upon this basis towards industrial applications.

1.5 – Demonstrative Example

This study will use a demonstrative example to contextualize the use of each package reviewed. The example of following a complex path, originally on a flat surface, will be used. This path is the point which the Tool Center Point (TCP) of the robot follows from path start to end. The TCP is a unique point and coordinate frame on the End Effector (EEF) of the robot. An EEF is a general term for any type end-of-arm tools, grippers, manipulators, etc. with which the robot interacts with the environment. More than one TCP may be defined per EEF.

A robot's path is a general term for the line which the TCP follows in space over the time of the robot motion. A planar path is a path which is constrained to a single geometric plane in a real 3-dimensional space ($\mathbb{R}^3$). Complex planar paths are defined in this study as any path which has more than one regular repeated geometric path or is entirely irregular and non-repeating. Figure 1 shows three different paths on a flat plane. The first path (A) is non-complex as it is comprised of a repeating, raster-like series of lines. Paths (B) and (C) are complex paths.

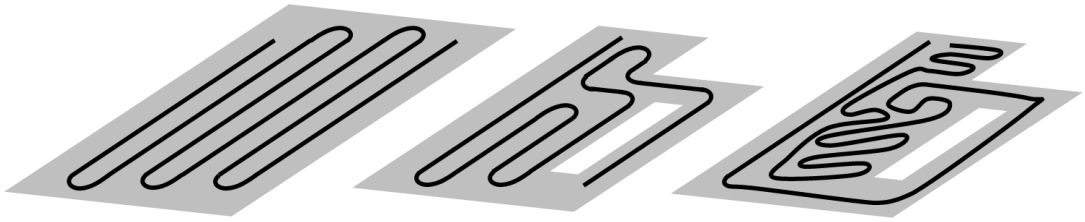

Figure 1: (A) Singular Regular Path, (B) Mixed Regular Path, (C) Oblique, Irregular Path

Creating these paths using a TP or ROS motion planner would require programming a pose at each critical point and the move type between each point along the path. Table 1 estimates the number of unique points required to program each path shown in Figure 1.

Table 1: Pose definitions required to define paths from Figure 1

Path	Critical Path Turning Points
A	12
B	18
C	Over 40

Complex path plans are useful when sanding, painting, machining, or welding with a robotic system. The majority of these applications will be assembled from a series of non-complex, regular paths to create a complex path. Few applications require a fully irregular, complex path, but some detailed painting paths may benefit from the ability.

A pixel grid maze was selected as the source of the demonstrative example. Mazes may be generated by drawing a white path on a black background and used to generate a unique, irregular path from which to derive a path plan. The paths shown in Figure 2 exhibit the same regular and irregular paths simplified onto a 10x10 pixel grid.

Figure 2: Gridded Maze Examples

Using an open-source maze-solver allows any maze with a start and end pixel defined to be solved and return a list of points along the path. A visualization of this is shown in Figure 3. A single white pixel along the top edge is defined as the path start and any single white pixel along the bottom edge as a path end option.

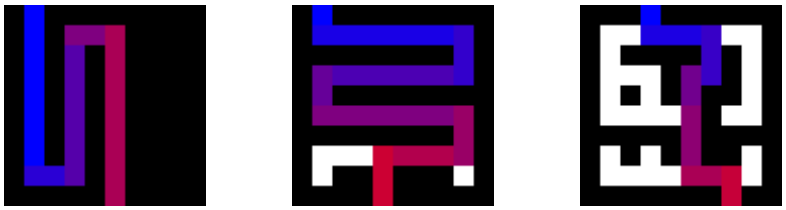

Figure 3: Gridded Maze Examples Solved Paths

A binary (black or white) pixel grid basis allows further expansion to irregular paths defined by the user (Figure 4) or another program. Another program may develop a path from a camera vision input. Through exporting this path as a binary image, a path may be found and passed onto a motion network.

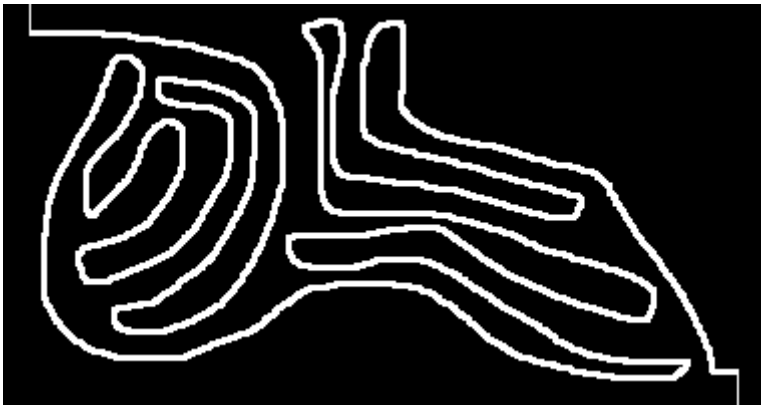

Figure 4: Irregular Path Created from Hand-Drawn User Input

The goal of this study is to assemble a motion planning and execution system using existing open-source packages. Further, to demonstrate these motion capabilities using a machine-vision defined path plan to instruct motion to the robotic system.

Chapter 2. Background

Successful deployment of a physical robot requires knowledge of mechanical design, kinematics, software design, and information networking. Integration with other sensors or manufacturing systems further expands the knowledge requirement of roboticists. This chapter will contextualize the information from these fields as relevant to this project or industrial robotics overall.

2.1 – Spatial Pose Descriptions using Transformations

A coordinate system is a broad term for conventions which allow events or objects to be located in space and time. A rigid body's pose, the combination of its position and orientation at one instance in time, may be described by a reference frame. This reference frame is a special coordinate system relative to the rigid object. Typically, this relative position is static in both space and time. Reference frames are just one kind of coordinate system, others, like the world frame, do not need to be attached to a body.

Any change in pose by a rigid body within n-dimensional Euclidian space ($\mathbb{R}^n$) may be described by an $(n+1) \times (n+1)$ matrix. Considering the real world to be an $n = 3$ dimensions world, objects are described using a $[4 \times 4]$ matrix known as a Homogenous Transformation Matrix.

Mathematically describing a translation requires only the relative change in position of a single point on the rigid body. Describing a rotation may be accomplished using numerous techniques. A rotation matrix is one such technique where rotation in n-dimensions is described by an $n \times n$ matrix.

A rotation of 90° about the Z-axis and a translation of $< +2\hat{x}, +3\hat{y}, -4\hat{z} >$ may be described as follows:

$$Rot(90°) = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$Translation = \begin{bmatrix} 2 \\ 3 \\ -4 \end{bmatrix}$$

Applied to a Transformation Matrix, T:

$$T_{template} = \begin{bmatrix} R_{11} & R_{21} & R_{31} & T_1 \\ R_{12} & R_{22} & R_{32} & T_2 \\ R_{13} & R_{23} & R_{33} & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 0 & -1 & 0 & 2 \\ 1 & 0 & 0 & 3 \\ 0 & 0 & 1 & -4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, the rotation and translation may be described by a single matrix. This leverages the mathematical flexibility and programmatic robustness of linear algebra. Further discussion on the mechanics and benefits of this technique may be found in the textbook, *Modern Robotics: Mechanics, Planning, and Control* [3].

2.2 – Robot Poses using Forward and Inverse Kinematics

All motion sent to the robot must be sent as joint positions. A joint position is a 6-element list detailing the absolute rotation of each joint in radians. Most industrial robotic arms use range limited, revolute servo motors. Individual servo ranges are typically less than 1 revolution in either direction $(-180°, +180°)$ and centered at zero. The ranges capable by a Motoman MS210 industrial robotic arm are given in Table 2.

Table 2: MS210 Motion Limits

Joint	Motion Limit [deg]	
	Min	Max
S	-180	180
L	-60	76
U	-86	90
R	-360	360
B	-125	125
T	-360	360

While possible to define a joint position and command robot motion to that position, it is an inconvenient method of pose definition when programming. Further, robotic programs are typically written from the context of placing the end of the robot, the end effector (EEF), at a specific Cartesian point and rotation relative to a global coordinate system.

Considering the 3-revolute, planar robot arm in Figure 5, the position may be defined using either method. Within each method, different coordinate systems and frames may be used to describe the pose of the robot. For example, in (B) the origin of the reference frame may be placed elsewhere in space to give different (x, y) values for the same robot pose.

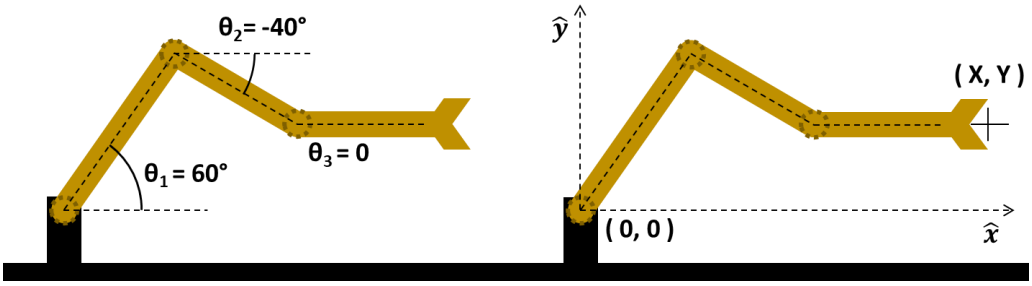

Figure 5: (A) Joint-Defined Position, (B) TCP Cartesian-Defined Position

A kinematic based method to convert this Cartesian point into joint positions of the robotic arm is needed. When the robot is moved to these joint positions, the EEF of the robotic arm should align with the desired Cartesian point. When applied to open-kinematic chains, this method is known as inverse kinematics.

Inverse Kinematics: Calculating the joint values of robotic arm from an EEF position.

Forward Kinematics: Calculating the position of an EEF from specific joint values.

Forward Kinematics (FK) is typically a closed-form mathematical operation where the use of geometric calculations will robustly solve the location of a kinematic chain. Inverse Kinematics (IK) methods may find no solution, multiple solutions, or an infinite number of solutions. For this reason, IK for high DOF robotic systems is an actively researched field where both analytical and numerical solution methods may be applied. Further information may be found in the textbook, *Modern Robotics: Mechanics, Planning, and Control* [3].

2.3 – Differentiating Motion Planning and Path Planning

In the context of organizing the roles of packages in ROS, a loose distinction can be made between “Path Planners” and “Motion Planners.” Path Planners are tools defined to automate or assist users in the generation of a path for the EEF to follow through space over time. A path is an ordered list of points in space that the robot EEF should move through. Motion Planners are advanced packages which deploy a variety of IK solvers to convert the Cartesian-defined path into a joint-defined trajectory. In ROS, motion planners often include collision environments and can amend a trajectory to move around a potential robot-environment collision.

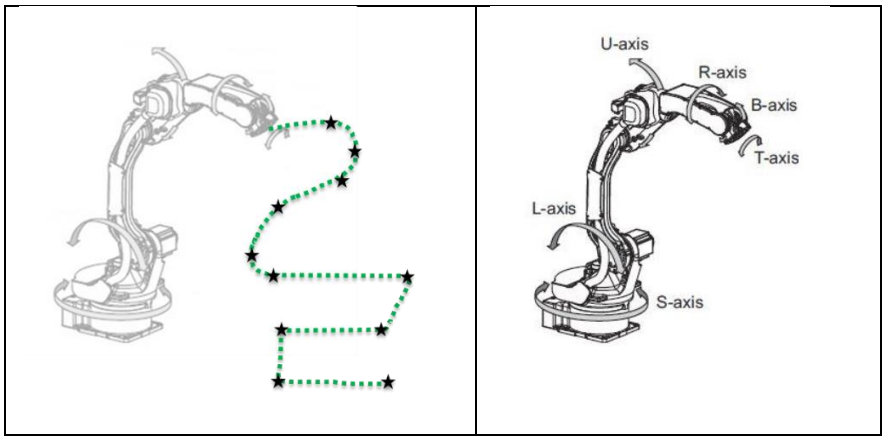

Figure 6: Path Planning vs Motion Planning

Motion planners often rely on other path-finding algorithms and libraries such as the Open Motion Planning Library (OMPL). OMPL is a software package containing multiple sampling-based path planning algorithms which are abstract to any environment, collision, or visualization tools. Motion planners, such as MoveIt, provide a ROS interface which builds upon these algorithms to generate motion.

2.4 – ROS Architecture Overview

When assembling a ROS powered robotic system, developers generally start with a rough package-type template. The role and relevant configuration details of each of these packages has been detailed here. All examples will be contextualized to the Yaskawa Motoman equipment used in this study's hardware demonstrations.

A: Controller Layer

Specific to most manufacturers' controller is an add-in like software which creates an interface between the ROS traffic received from a TCP/IP network and the OEM's control drivers. Yaskawa Motoman produces the "Motoman ROS Server" written in MotoPlus as open-source code developers may compile. Precompiled binaries tailored to each controller are offered: FS100, DX100, DX200, YRC1000, YRC1000micro, and collaborative robot variants. This code is installed directly onto the controller.

OEM's additionally offer a software driver package which users include in their development workspace to define and allow communication between the local ROS environment and the controller-plugin. Yaskawa Motoman uses the "motoman_driver" which pairs the community generated robot models with the MotoPlus plugin on the physical controller.

B: Controller Messages – “simple_message”

Being a core design intent of ROS, command and response communication between the local driver (on a computer) and the hardware controller are standardized to common messaging rules. Motoman uses a family of messages known as “simple_message” and includes the following messages:

- Joint trajectory (command)
- Joint feedback (response)
- Read I/O Message
- Write I/O Message

Joint messages use the radial position values each of the 6-joints (SLURBT). Input/Output (I/O) messages use the internal hardware address.

C: Robot Support Packages – The Interface layer

Robot support packages describe the physical robot and provide ROS-specific launch files. The robot description includes a URDF (universal robot description format) and digital CAD models. URDF’s define the location of each joint, limit of rotation, link orientation, and – optionally – kinematics. Launch files use the OEM’s drivers and robot description to establish a connection between the ROS environment and the physical controller. All motion and I/O control commands are sent through this connection.

In ROS-Industrial, the structure and naming of these packages is standardized across all OEM’s. Properly deployed with the OEM’s driver, users may directly (by terminal commands) or programmatically move the robot and controller end effectors.

D: Motion Planners

Motion planners are tools which simplify the joint-type motion planning process. Users input the start and end position of the end effector and the motion planner determines the necessary motions the robot needs to follow to move between these points. For a wheeled robot, like a small room-sweeping robot, this is a simple process of figuring out how to move the drive wheels such the robot stops in the correct location. A 6-axis robotic arm is a more complex process and requires the use of inverse-kinematics. Motion planners cleanly wrap these mathematical solvers to a simple interface.

Different flavors of path planners allow the motion between points to be tailored to the needs of a project. Pick-and-place systems (ex. a part sorting robot) only care that the part ends in the correct place. Whereas both the start/end points and path of a CNC milling machine are critical. Specific ROS planning packages will be reviewed in Section 3.2 – Review of Existing Motion Planners.

E: Path Planners

Path planners enable users to automate creation of (1) the desired path of a TCP and (2) more complex paths to be constrained. Highly repetitive paths such as the parallel paths found in painting or machining are well suited to this as a path planner may generate a raster-like path along the surface. Geometrically complex paths are also needed in tasks like surface inspection, navigating highly curved surfaces, or welding applications.

This final planning capability enables developers to create automated and autonomous solutions through the use of ‘smart’ path generation tools. Existing path planners available within the ROS Industrial community are detailed further in Section 3.3 – Evaluation of Existing Path Planners.

2.5 – Advantages and Impact of ROS

The flexibility of the ROS architecture removes significant barriers to research and development. Industry application developers may assemble packages into a custom deployment for each new project with minimal reprogramming and setup work. Researchers may use this flexibility to focus time on the development of new capabilities instead of rebuilding specific test scenarios. Further, developing programs and tools within the ROS framework ensures that all are abstracted from hardware and able to be used with any family of robots, regardless of the manufacturer – a capability unprecedented in the robotics field. Further research and development into the ROS framework enable the next generation of robotics on the manufacturing floor.

2.6 – Assembling a Robotic Network for Motion Planning

Robust motion planning is achieved through the integration of the robot model, collision checkers, and environment awareness to inform the motion planner. Early software stacks like ROS’s “motion_planning” showed an early attempt at integration of these components. As the field matured, the software stack evolved and new packages were developed offering specialization towards different types of motion.

The final stage of developing a robotic motion plan is defining the series of key points the tip of the robot should navigate through. This information is known as the "path plan." Path plans may be manually defined, similar to the way motion is programmed on a generic each pendant. But the advantage of ROS is the ability to algorithmically generate a new path plan during program execution. Examples of a unique path plan exist in machine vision driven pick and place programs. The camera identifies an object in space and the path planner uses this location to generate a path to move the robot gripper from its current position to the object. Passing the new path plan to the motion planner, the software stack will determine how to move the physical robot joints in synchronous to follow the path plan.

Chapter 3. Methodology & Physical Implementation

Common practices, package requirements, and lessons learned in assembling a ROS-based motion planning and control environment are detailed in this chapter.

3.1 – The ROS Environment

Prior to this study, extensive development efforts were performed to standup the ROS-Industrial configuration and controller interface packages to allow simulation and physical testing of the planner packages. As this study was the first ROS project in the AIMS Lab, each system had to be configured for a first-time only setup. The work performed supports this study and all future ROS projects in the AIMS lab.

As a loosely connected, package-based framework, ROS utilizes a single "Master" to manage the storage of environmental variables, register active nodes (tools or programs), and maintain active topics (message pipes). A standard configuration of ROS minimally includes the code modules to create a ROS Master and the subsequent environment. All descriptions of robot packages in this study will be intended for Yaskawa Motoman robots, but should be generalizable to all robots.

Deploying a robot into the ROS environment requires the robot support and driver packages. Together these packages may be launched to setup the external connection with the physical robot controller. At this stage, the system is equipped only to turn the

servo motors off/on, control IO ports, and send joint motion commands to the robot. Additional details on the environment setup are detailed in Appendix B. Lab Specific Environment Setup Information.

3.2 – Review of Existing Motion Planners

Motion planners convert a path plan (the series of key points along a TCP's trajectory) into actual robot motion while considering various restrictions such environment collisions, self-collisions, actual reach length, and kinematic singularities. Three motion planners were identified and reviewed: MoveIt, Descartes, and Tesseract.

MoveIt is a Motion Planning Framework which includes a motion planner (move_group) powered by a variety of planners such as OMPL, STOMP, SBPL, and CHOMP. MoveIt also provides visualization plugins, planning scenes, collision detection, and more as a multi-use tool in robotics [4]. MoveIt is designed for free space motion planning where each motion is joint-optimized. Such optimization attempts to find the path which moves each individual joint through the shortest change. This often results in a non-straight path for the EEF, but makes it ideal for pick-and-place motion tasks where only start and end positions are critical. Further, MoveIt has evolved to implement basic Cartesian path planning capabilities, but has been unable to meet the same capabilities of other specialized packages.

Early motivations to develop a Cartesian path planner based in Moveit are documented in the ROS-Industrial "ROS Enhancement Proposal" REP-I0003 [5] and within the following year the Descartes project was initiated. Further development has continued on other planners and techniques as highlighted in Picknick's 2021 summary white paper [6]. Two cartesian, semi-constrained motion planners were reviewed in this study as they have been thoroughly tested and deployed by the industrial community.

Descartes began development in 2014 with support from NIST and the ROS-Industrial Consortium Americas to development a package capable of semi-constrained (5 DOF) Cartesian paths. Descartes is a hybrid graph-based/interpolated planning package [7]. Path plans in Descartes are able to be defined using multiple types of definitions: joint-defined positions, Cartesian-defined positions, and other specialized methods. Each trajectory key-point may be defined with a tolerance of how 'close' the robot need to get to that point. This is visualized in Figure 7 where the size of each circle represents the tolerance. In this aspect, the planner functions similarly to a traditional Motoman teach pendant program where a mix of Linear or Joint path types, with tolerances, may be combined into a complete trajectory.

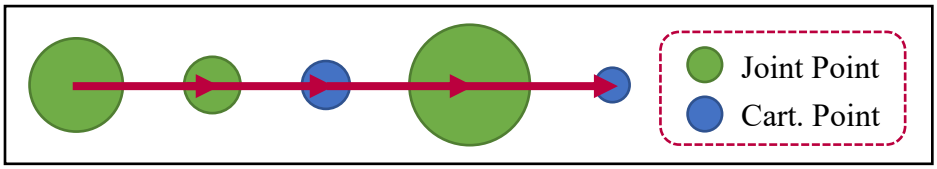

Figure 7: Hybrid Trajectory with Joint-defined and Cartesian-defined points

Tesseract it a light-weight planning framework which is ROS-agnostic with full C++ and Python support and inherently uses convex hull collision checking. It is purpose built for the TrajOpt trajectory planning and optimization library. TrajOpt was benchmarked to solve faster than OMPL and CHOMP planners for various DOF examples [8] as detailed in Table 3.

Table 3: Results on 198-robotic arm planning problems [8]

Simulation of a PR2 robot's arm (7-DOF) concluded TrajOpt as the fastest algorithm. Methodology did not use Tesseract framework.

Metric	TrajOpt	OMPL-RTTConnect	CHOMP-HMC
Success Fraction	0.818	0.854	0.652
Mean Solve Time (s)	0.191	0.615	4.91
Mean Normed Length	1.16	1.56	2.04

The advantage of Tesseract/TrajOpt is the inherent use of a collision environment when planning and the ability to build different types of trajectories. This combines the advantages of MoveIt and Descartes highlighted earlier through allowing deployed use cases like those detailed here:

- Fully Constrained Cartesian Path
- Semi-Constrained Cartesian Path
- Free Space Path
- Semi-Constrained Free Space Path
- Free Space + Constrained Cartesian Path

In reviewing the reported capabilities of each of these planning packages, a concerted effort was made to deploy each into a simulated and real environment. Only the MoveIt framework was successfully built and tested, whereas Tesseract and Descartes could not be deployed due to numerous dependency and configuration issues. The MoveIt package was selected for use to continue developing the sample application in this study.

3.3 – Evaluation of Existing Path Planners

Path planners automate or assist users in the generation of an end effector (EEF) path. A path is an ordered list of points in space that the robot EEF should move through. These types of packages are generally specialized to different types of projects: sanding, painting, edge following, etc. Not all projects require a dedicated path planner. The traditional pick-and-place project does not require a path planner as the motion between points is not critical to the operation.

Few standalone path planning packages exist in the ROS Industrial community. This study identified the Noether and Bezier packages as automatic path planners available to plan paths on 3D surfaces. During the review of these packages, neither was deployed due to errors that were unable to be resolved in the limited time available.

Noether [9]

A SwRI developed package designed to generate tool path plans using user input mesh files. Figure 8 demonstrates the two types of path plans offered: surface raster plans and boundary edge path plans. This package is hosted by the ROS-Industrial Consortium.

Surface Raster Path *Mesh Boundary Edge Path*

Figure 8: Path plans generated by SwRI's Noether package

Bezier [10]

Developed by the Institut Maupertuis, this package addressed a need to automatically generate grinding tools paths for 6-axis robots across an input mesh file. The generated robot poses form a rectilinear trajectory similar to the Noether package and includes a capability to "dilate them in all directions in order to grind defects with a pass principle" [10]. A capability demonstration is provided by the package authors and is shown in Figure 9. Bezier is hosted by ROS-Industrial.

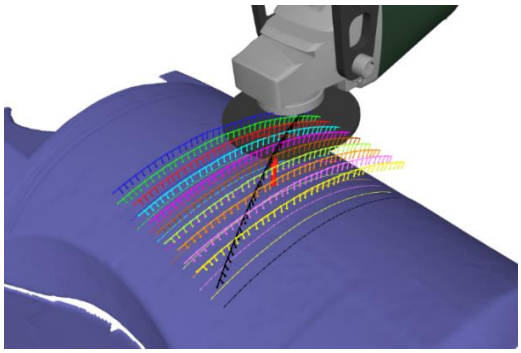

Figure 9: Bezier's rectilinear trajectory across a previously unknown mesh file [10]

3.4 – ‘MazeRunner’ Package Development and Structure

This study developed package, “MazeRunner” to generate a path plan in which to execute motion via a Motoman MS210 industrial robot. Within the ROS-Industrial Consortium’s open-source code, support packages for many of the Yaskawa Motoman robots have already been created. These packages are supported and approved by Yaskawa engineers to ensure they are sufficiently robust for industry deployment.

1 – Path Processing Tool

Using the grid-based maze image as an input, an open-source maze-solving package was used to solve the path. This path was then exported as a CSV file listing each node along the resultant math. A node sets a point along the path, but does not define the shape of the path between nodes. At this stage, the path nodes are defined as (x, y) pixel coordinates relative to the top left pixel of the input graphic. This process is visualized in Figure 10.

To use these points in our real world, $\mathbb{R}^3$ space, a third dimension was added with value zero. The resultant path is now a list of (x, y, z) points where $z = 0$.

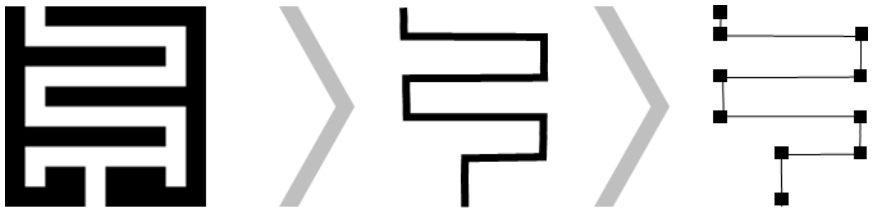

Figure 10: Extracting Path Nodes from a Pixel Grid

2 – Transforming Paths

An object class was created to facilitate the usage of homogenous transformation matrices with linear algebraic calculations throughout the python package. This tool deploys the concepts outlined in Chapter 2.

The initial function converts each node in the path to a homogenous transformation with an identify rotation. Thus, each node along the path is able to later be modified individually to represent a unique orientation – a critical capability for curved surface navigation.

Each node along the path is defined relative to origin of the maze. In terms of reference frames, each node is its own unique reference frame relative to the body frame of the maze. The origin of the maze is always the top-left pixel corner of the maze following image processing convention.

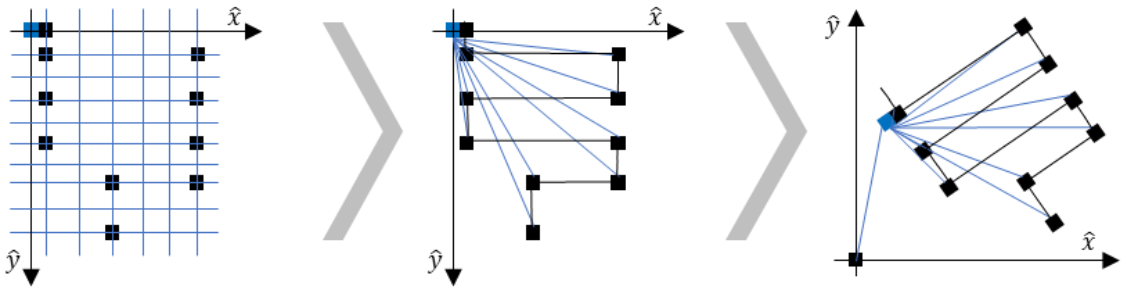

Figure 11: Transformations Applied to Path

Using the concept of reference frames, the maze may be considered a rigid body with a frame defined at the origin (top-left corner). Thus, each point along the maze path may be defined as a pose with a transform relative to the maze's reference frame. When the maze is placed into a world environment, a transformation may be found to describe

the position of the maze body (and subsequently the maze path) relative to the world (fixed) reference frame. This capability simplifies image processing as only the pixels contained by the body of the maze need processed by the path finder (ie. maze solver).

3 – Visualizations Tool

A simple tool was developed to allow developers to visualize imported points in a 3-dimensional plot. This simple tool was developed on top of the python plotting package, matplotlib. This is useful for visually confirming the node path and checking the effects of transformations upon the maze. This result is plotted in Figure 12, note the origin of the maze body is coincident with the plotted coordinate system. This origin is plotted as a blue circle at point $(0,0,0)$.

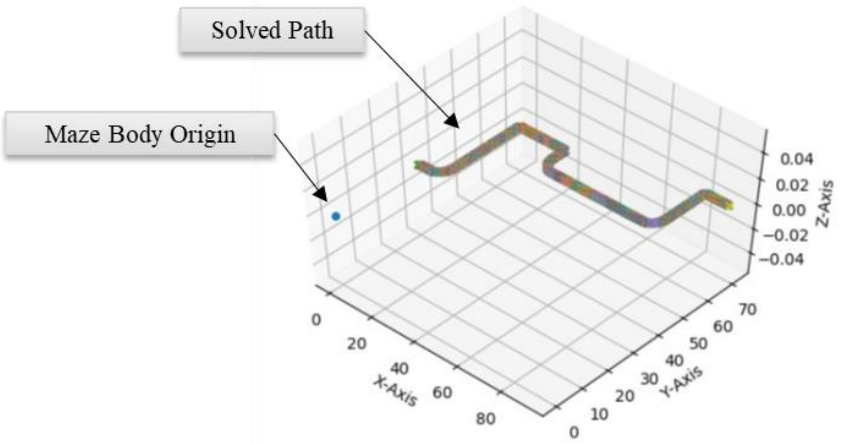

Figure 12: 3D Plot of Maze

4 – Machine Vision

Machine vision was utilized to determine the pose of the maze body and to generate an image from which the maze path could be solved by a later program. To uniquely orient the maze, a simple 3-dot locator system was added to the maze presented to the robotic cell. Vision was possible through the use of an Intel RealSense D435i camera. With stereo optic sensors, RGB camera, and an IR projector, a full featured data stream may be provided to the program. Intel produces the RealSense library and a variety of wrappers to allow developers to connect with the camera.

When setting up a vision pipeline to the Intel RealSense camera, multiple parameters must be configured first. The desired sensors must be configured with a requested resolution, frame rate, data format, and (for depth) units. Depth streams may also be limited by maximum distance thresholds if the distance from the camera to the region of interest is known. For the maze-runner demonstration, there is no minimum distance and the maximum distance was set to 6 meters; roughly twice the reach of the robot and equivalent to the ceiling height of the surrounding environment.

The onboard Intel D4 Vision Processor offers onboard RGB & Depth alignment and real-time vision filtering. Optional onboard processing reduces the computation overhead required for developers and helps standardize multi-camera systems. Both the alignment and filtering capabilities were deployed in this project. Alignment is critical to allow accurate length and distance measurements based upon the data streams. Figure 13 illustrates the need to align a common reference frame.

The origin of the camera's global reference frame is located internal to the camera body behind the left stereo camera (2nd from right as show in Figure 13). Stereo optic-based depth measurements are reported from the perspective of the camera frame. RGB images are collected from the left-most sensor (right-most sensor in Figure 13). Prior to taking measurements from the RGB images the data must be aligned with the depth data and transformed to the camera's coordinate system.

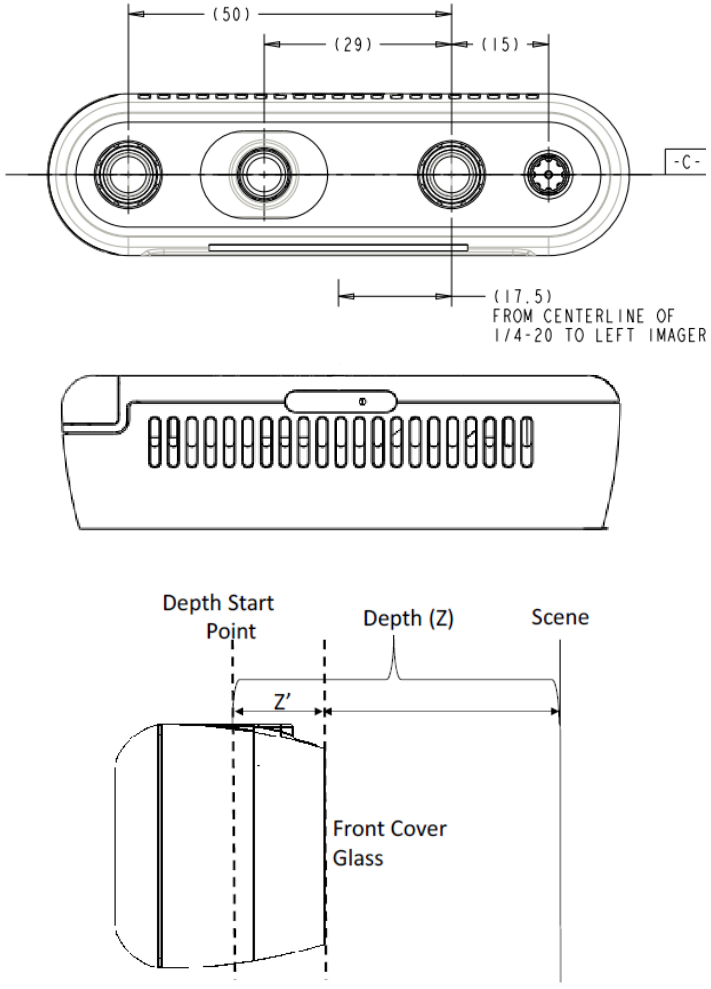

Figure 13: Intel RealSense D345i Reference Frames [mm] [11]

Integration of the camera into a robot's URDF must be done with the recognition that the reference frame's origin is internal to the camera. Figure 13 shows the depth along the z-axis of the camera to be 4.2 mm as reported by the product datasheet [11]. This understanding is applied to the design of the demonstration end effector detailed in Appendix A. End Effector Design.

Four types of on-board depth post-processing for the depth stream are provided [12]. Correctly applied, these filters enable the resultant data stream to be used directly without further processing.

Decimation Filter: Scale image size using $(n \times n)$ median depth value

Spatial Edge-Preserving Filter: Smoothing of depth data

Temporal Filter: Depth data persistency on a historical frame basis

Hole Filling filter: Wrapper for multiple hole-filling techniques

Based upon parameters suggested in the product documentation and empirical testing, the spatial edge-preserving filter was applied to reduce anomalies in the depth cloud which will be later used in picking individual pixels as the identifying maze locators.

The D435i pipeline opens multiple data streams to the user depending on the setup and wrapper utilized. Only one pipeline may be opened to a camera at once. Two application wrappers for the RealSense Library were evaluated for this study: Python and ROS. Wrappers provide tools which allow scripts to access the camera vision and prepare the data for use downstream in the software stack. Both wrappers use the same basic setup outlined in Figure 14.

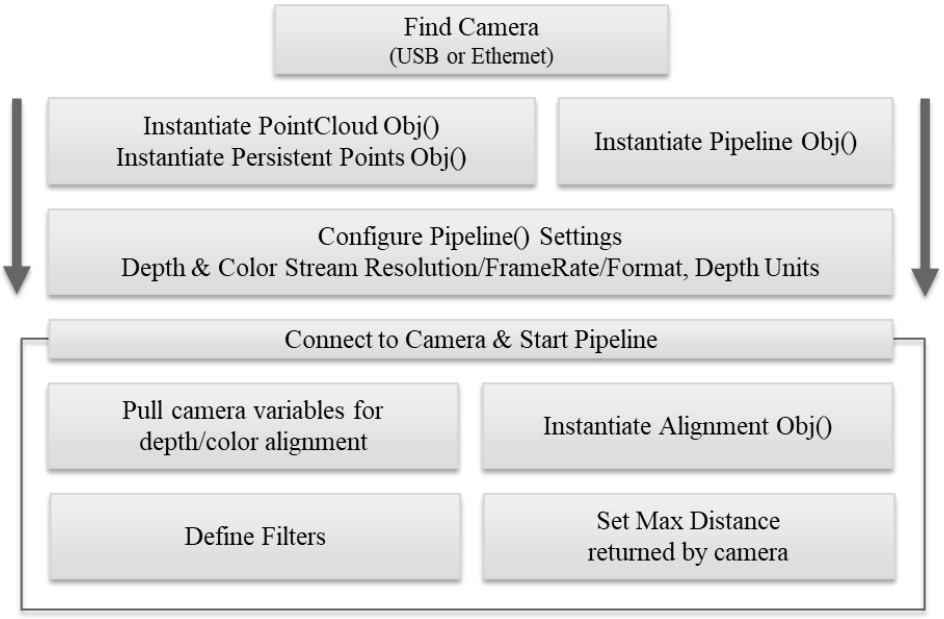

Figure 14: Basic RealSense Camera Pipeline Setup Steps

Python Wrapper

Intel provides a Python wrapper for the RealSense SDK with numerous examples to setup the cameras and begin capturing data streams. Python documentation and examples are not as comprehensive as for the C++ wrapper, but all RealSense components are exposed within the Python wrapper.

Using the wrappers objects and function calls users may instantiate, configure, and launch a pipeline with RealSense cameras. This wrapper allows “Advanced Settings” to be modified and used to calibrate the camera. On-camera filtering may be modified and enabled when launching the pipeline.

Once a pipeline is established, it is possible to capture single frames from the camera’s color and depth data streams. Through using Python, the user has direct control of the frames and extensive flexibility on how to process each frame.

Some options relevant to this study include:

- Single frame alignment
- Secondary frame filtering
- Color image masking (remove pixels which exceed max depth in depth frame)
- Export Data (png, Numpy array data, ply mesh files, point cloud files)

Figure 15 shows a colorized rendering of the depth cloud as exported to a PLY mesh file. Such renderings are useful to help visualize and understand the effect of filters by looking at a static frame.

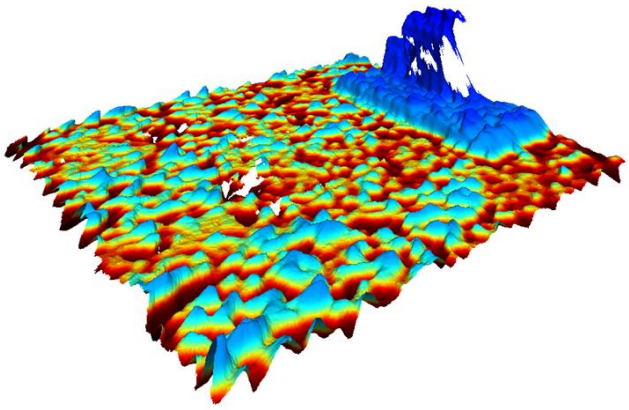

Figure 15: Colorized Depth Data

To determine the coordinate of a frame point in world units (non-pixels), the camera's internal intrinsics (or calibrated parameters) may be used to calculate the XYZ position of a single pixel relative to the camera frame. In general, camera intrinsics describe the focal length, principal point, pixel width and height, and lens distortion. Following the Intel Realsense documentation, this information is used to determine the real-world coordinate position of pixels relative to the camera reference frame.

ROS Wrapper [13]

Intel provides a simple ROS wrapper (realsense2_camera) to demonstrate the capabilities of the RealSense library in the ROS Melodic and Noetic environments. Out of the box, the wrapper provides an XML-based launch system to connect to a USB connected camera and setup a pipeline with control over nearly all pipeline parameters. Limitations exist in specifying the input parameters to each filter, but it is possible to select which filters should be enabled. Once the pipeline is established, the wrapper package can publish an extensive, customizable list of topics to the ROS environment. These topics an include raw, aligned, filtered, and otherwise modified versions of the camera's color and depth data streams. An example of a sucessfully launched camera produces a topic graphic similar to that shown in Figure 16.

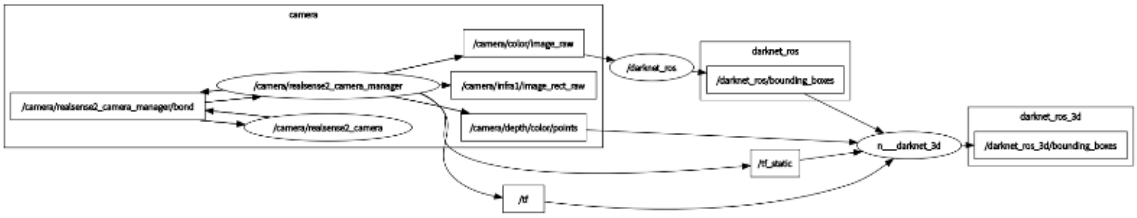

Figure 16: RealSense Published Topics visualized by RQT's Introspective TopicGraph

These capabilities allow users to rapidly visualize their data stream in RViz using existing ROS plugins for point clouds, depth streams, and video streams. Further, depth data may further be leveraged through registering the position of the camera with ROS's transform tree (TF package).

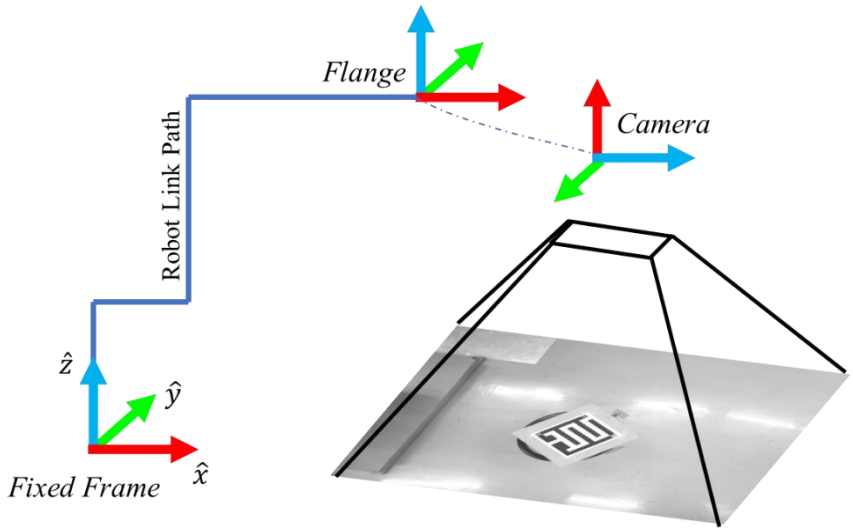

Figure 17: Connecting transformations to find camera points in the World (Fixed) Frame

This allows each depth or point cloud point's position to be reported relative to the global coordinate system. Knowing these points true position in near-real time allows the data stream to input to a collision environment or inform a path planner. Figure 18 shows the same scene taken from similar angles in both the real world and the simulated environment with depth & color camera data streams included.

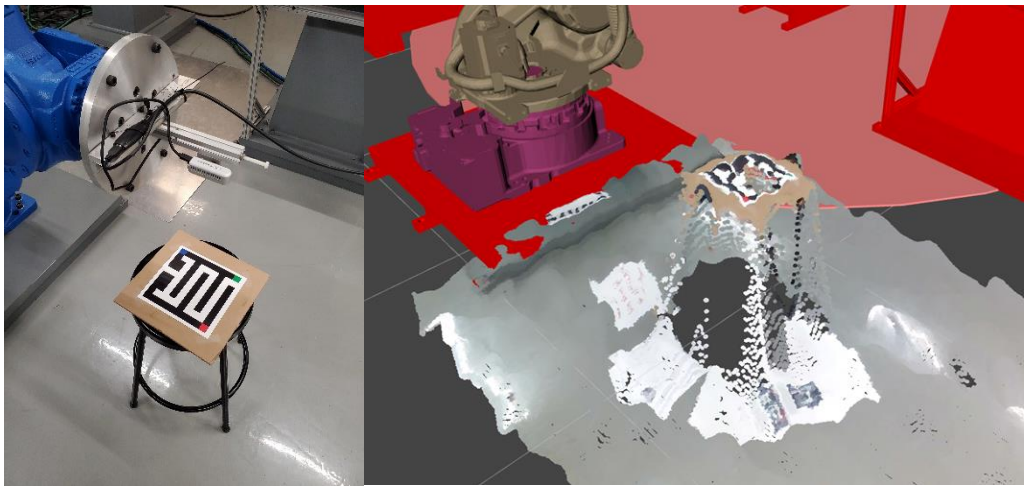

Figure 18: Depth Stream Data shown in a simulated environment reflecting

The Python wrapper was selected for use because of the ability to capture singular frames and ease of export to save external to ROS. This capability was essential to allow post-capture vision filtering.

5 – Machine Vision Filtering

The static data collected from the vision system must be filtered to find key points in the image. The goal of this filtering is to identify the location and size of each of the colored locating points on the maze. DREAM.3D is a data analysis tool to rapidly assemble image processing pipelines from a wide variety of filters [14]. The advantage of DREAM.3D is that the GUI based, filter assembly tool is a convenience tool for building the pipeline (stored in the JSON format). The actual pipeline is called from a terminal and is processed separately from the current thread.

An important recognition in processing image data is the representation of color and depth frames as projections of a pixel grid. The origin of both image and depth data are at the top left of the original image. To simplify finding the position of each pixel relative to the camera, an internal RealSense function call is used to return the position of a pixel. This method accounts for the camera intrinsics, pixel scale, and alignment of the color and depth frames automatically. Figure 19 represents the coordinate system used and reality that each box, or pixel, represents a depth value projected away from the camera lens.

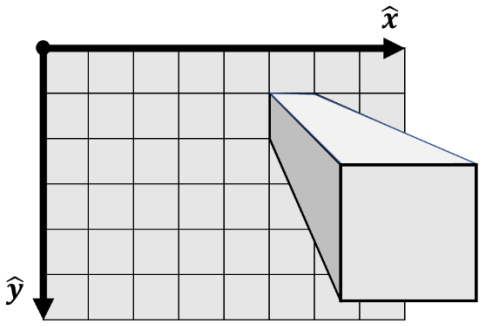

Figure 19: Image Coordinate System for a Depth Frame

Within DREAM.3D, attribute arrays are the primary method of data storage and may be thought of as multi-dimensional matrixes. Numerous filtering techniques were attempted and the following was found as the most robust. Table 4 details images from this process.

1. Import RGB Image

   The "ITK:: Image Reader" is used to load an image into a 2-dimensional attribute array where each cell is a 3-element list of the (red, green, blue) content of each pixel.

2. Separate Image into 3 grayscale arrays; R, G, B Filter, "Split Multicomponent Attribute Array," generates three new 2-dimensional attribute arrays. Each array represents a single grayscale image where each cell is a value from 0 to 255 of the brightness of that color at that pixel.

3. Threshold: Masking Colors

   Using a "Threshold Objects" filter, masks were created for colors white, red, green, and blue. These colors are the regions of the maze path (white) and the locator dots (red, green, blue). The mask is an attribute array of binary values (either 0s or 1s).

Therefore, the ‘red’ mask array will have a ‘1’ value for each cell which aligns with a ‘red’ pixel on the original image.

4. Filter: Dilation of Mask

   An “Erode/Dilate Mask” filter was used to dilate each R,G,B colored mask three times. In each dilation, the surrounding cells of the original “1” valued cell were turned into a “1” value as well. This increased the size of the mask and helps combine regions of the same color in real life that the threshold technique incorrectly split.

5. Segment Features (Scalar)

   For each of the four masks (the maze and locator dots), filter “Segment Features (Scalar)” is used to group pixels of a similar coloring within a scalar tolerance. Each unique group is assigned a unique feature ID and a new attribute array is created to store this data.

6. Find Feature Sizes and Find Feature Centroids

   The “Find Feature Sizes” filter inspects each group of pixels found of each color given and calculates the volume of each pixel group (number of pixels contained in a group). The “Find Feature Centroids” filter finds the center of each group of pixels.

7. Export Feature Data as CSV File

   The CSV export command is used to output the collected location and size of each dot in the array. This allows the collected data to be saved for use in later scripts.

Additional filters within the pipeline, such as the “Write DREAM.3D Data File”, allows users to automatically export all data generated into an XDMF file for debugging.

Table 4: DREAM.3D Image Filtering

Filter Step	Image
Import RGB Image	
RGB Separation	*Red* *Green* *Blue* 
Color Thresholding (to find locator dots)	
Path Detection	

Once the pipeline has been built, it may be executed by calling the pipeline runner and inputting the pipeline (JSON) file. In Python, this may look like the following:

```
$ subprocess.call(["/opt/dream3d/bin/PipelineRunner", "-p", pipeline])
```

The disadvantage of this design is the inability to pass the results directly from the pipeline into the Python environment. Therefore, the pipeline is setup to save a unique file for each filtered color into a separate CSV file. These files may be then be input to the machine vision post-processor described in the next section.

6 – Machine Vision Processing

The vision processing tool imports the location and size of the dots identified by the DREAM.3D pipeline. Subsequent functions characterize the maze using the location of each of these dots. The initial application only uses the RGB image data to characterize the planar projected space. Future development will leverage the already aligned RGB data with depth cloud data to enable 3-dimensional rotation and centroid calculations.

Locators. The centroid of each of the three locator dots its loaded from CSV files exported by the DREAM.3D image filtering pipeline Once imported, the largest group of pixels for each color filtered (R,G,B) is flagged as the locator dot. This assumption makes the program susceptible to errors if a larger group of pixels is incorrectly flagged as the locator dot. While not implemented, a potential fix to this is including a conditional flag where the user is notified to manually select the correct pixel group if too many groups of pixels over a threshold size are found.

Planar Rotation. Using the three locator dots, we may determine the planar rotation on the camera's focal plane. The focal plane is parallel to the face of the camera lens and perpendicular to the camera's z-axis (depth). A 3-dimensional rotation matrix was derived from the normalized vectors representing each coordinate axis.

Axis Unit Vectors

$$x-axis = \langle x_x \quad x_y \quad x_z \rangle$$
$$y-axis = \langle y_x \quad y_y \quad y_z \rangle$$
$$z-axis = \langle z_x \quad z_y \quad z_z \rangle$$

$$Rotation\ Matrix = \begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix}$$

As the rotation is planar, the z-axis is defined as $\langle z_x \quad z_y \quad z_z \rangle = \langle 0 \quad 0 \quad 1 \rangle$. Further, the Euler rotation angle γ about the z-axis (ie the rotation in the focal plane) may be found using the following relationship.

$$Rot_z(\gamma) = \begin{bmatrix} cos\,\gamma & -\sin\gamma & 0 \\ sin\,\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

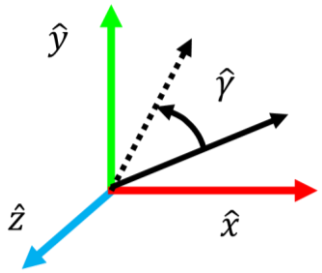

Figure 20: Rotation about Z-Axis

Maze Centroid. Given the assumption that projected region boundary is either square or rectangle, we may use the location of three of the four bounding corners to calculate the center of the region.

Maze Mask. To prepare the image for the maze solver, all regions except the maze must be masked out such that only the path region remains as white colored pixels. Two techniques were considered for this capability, the first was deployed:

A) 2-D Rotation from projected region:

Using the pixel location of the colored dots and the body rotation, the Python library OpenCV was used to generate a polygon defined by the four corners of the maze. This polygon was used to create a mask where the area outside the polygon was blocked. The mask was projected onto the DREAM.3D filtered path image to return an image of just the path. Finally, the resultant image was counter-rotated and cropped to just the maze region. This final image may then be input to the path solving program to generate an (x, y) path relative to the maze body frame.

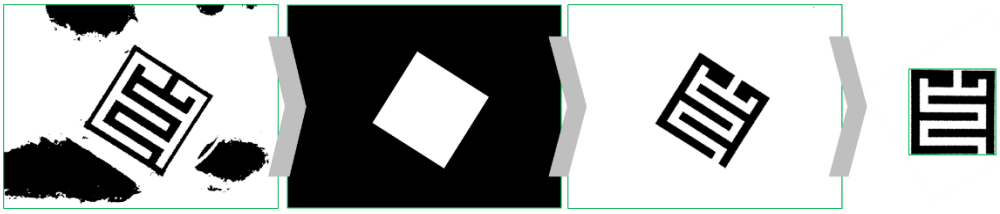

Figure 21: Rotating and Cropping Maze for Maze Solving Program

B) 2-D Rotation of Image

Using the previously calculated rotation of the maze, the entire image could be counter-rotated such that the coordinate system of the maze's body frame and the image's pixel coordinate system align. Two approaches were considered to find the key locator dots on the new image.

- Use geometric relations to calculate to find new centroids locations based on the rotation about a new point. Determine how full image has scaled relative to original and account for transition in calculations.
- Export the rotated image to DREAM.3D. Write a pipeline to re-identify each locator and export the new coordinates.

While neither method was deployed, the first method was evaluated to be sensitive to the scaling of the resultant image as documentation on the technique by Scipy and Numpy is insufficient. The second method was evaluated to be robust, but less precise as the re-filtering of the final image would may cause the centroid to walk relative the true center of the square.

Scale. Given an expected size of the maze, the pixel-to-meters scale may be calculated for later use. The length of the two sides of the maze (lines between each dot pair) was found in pixels and compared to the user-input length of the maze.

Transforming Identified Points to World Coordinates. Leveraging ROS's TF2 package loaded with the robot's URDF, the transformation from the world frame to the camera frame may be published onto a new topic. This is accomplished by creating a ROS node which continually publishes the desired transform to a topic. To access this transformation, a subscriber package may be written which reads the latest transform from the new topic and writes the transform to a CSV file. Therefore, whenever the

transform is needed the user may call the subscriber from a Python subprocess() to run once and export the desired transform as an XYZ position and quaternion rotation.

Loading the resultant transform and converting from quaternion to a transformation matrix, the camera position is now known in world coordinates. The body frame of the maze known relative to the camera can be found in the world coordinates by applying the newly found camera transformation.

7 – Planning with the Moveit Interface

A MoveIt configuration was developed and tested to allow use of the MoveIt framework in this study. This configuration loads the robot and end effector URDF into the framework. Further, the program was configured following examples in the ROS Industrial Motoman robotics community to connect with the Motoman ROS driver package enabling physical robot control and testing. Further details of this configuration are given in the Appendixes. Having established a functional testing network, a simple program was developed to loop through the list of robot poses forming the robot path.

3.5 – Maze Path Solver

Existing maze solving packages were used to process the masked maze from an input image into a sequence of (x, y) nodes forming a path. This solving package utilized Dijkstra to find a path between a given start (top white pixel) and end (any bottom white

pixel) point in the image. Further details on this package are detailed in this paper's associated Github repository: https://github.com/osu-aims/maze-runner.

3.6 – System Design & Assembly

Combining each of these components together, a basic level of automation may be achieved. The maze is presented to the robot and the control program is run to execute the steps outlined in Figure 22. After launching the robot connection, the automated motion input may be initiated using a roslaunch command.

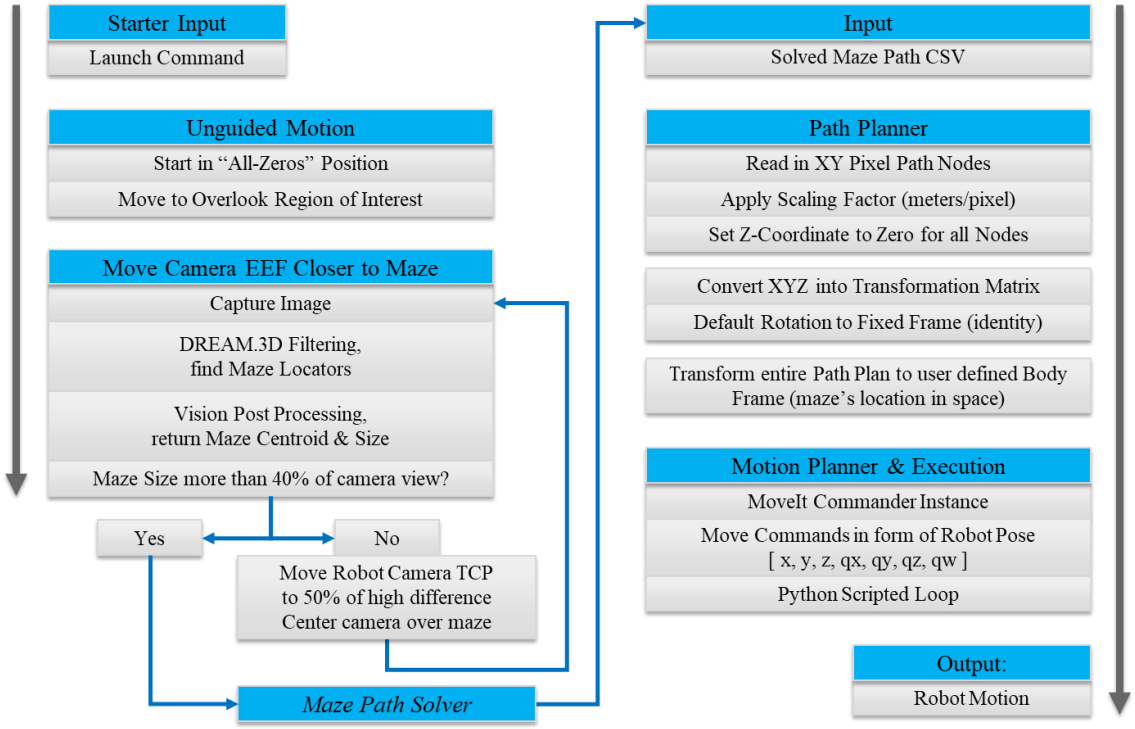

Figure 22: Integrated System Operations Flow Chart

Chapter 4. Initial Demonstration Results

4.1 – ROS Environment Testing

Evaluation was performed incrementally to verify each package and derived capability. These steps may be summarized as follows:

1. Robot Model: Simulated Model Accurate
2. Robot Control: Able to send motion commands and see action response
3. Robot Model: Directionality. Directional joint motion matches physical action
4. MoveIt Config: Able to Plan & Execute Motion
5. Vision: RealSense ROS Topics being published and visualized with Rviz
6. Vision: Single frame captured and exported images, point cloud inspected

Robot and EEF description models (URDFs) were verified empirically by navigating the robot using the MoveIt Commander such that the Pointer TCP would be touching the ground in the simulated model. As shown in Figure 23, the resulting pose placed the Pointer TCP 0.5 [inch] or 0.027 [m] above the correct position. This accuracy is acceptable for this test, but identifies a need to further improve the URDF.

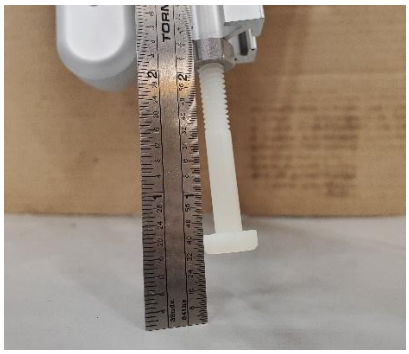

Figure 23: Robot and EEF URDF Model, Emperical Verification

Having successfully configured the ROS environment and connected to hardware, a series of path following demonstrations were used to benchmark the level of capabilities achieved from the current setup.

1. Pre-Solved Maze in Known Position
2. Unsolved Maze in Unknown Position

Details of each test are given in the following sections with their subsequent analysis.

4.2 – Pre-Solved Maze in Known Position

A maze was presented to the robot where the maze path had already been solved and stored in a CSV file. This solve path and the position of the maze body frame in the world coordinates were hard coded into the navigation system for testing.

Given this information, the robot successfully took an ‘overlook’ photo of the scene including the maze from the camera TCP’s perspective and navigated the pointer EEF through the maze.

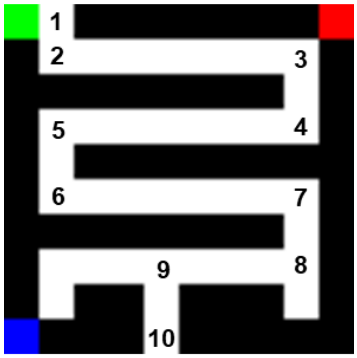

Figure 24: Nodes along Maze Path Plan

The path plan was correctly mapped onto the world environment and confirmed with the machine vision input to RViz.

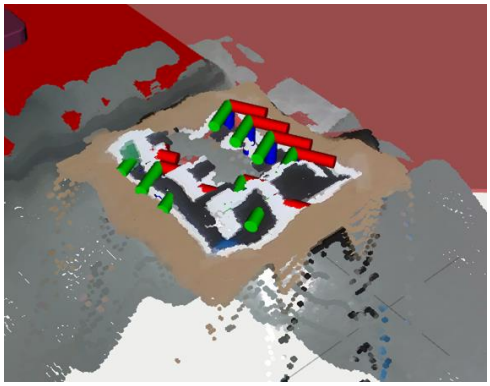

Figure 25: Path Plan Correctly Mapped onto World Coordinates

The path planner successfully navigated through the full maze 4 of 10 times. 40% of the motion plans failed to navigate to at least one of the 10 path nodes and skipped it to finish the path plan. The root cause of this is the usage of MoveIt; which was designed as a free-space, joint-optimized motion planner. Internal to MoveIt, this failure may be addressed by increasing the allowable planning attempts and planning time to ensure a solution is found each time. A more robust solution would be to switch to a motion planner optimized for this type of planar motion such as Descartes or Tesseract in the future.

Additionally, the planner's inverse kinematics will find solutions which result in excess motion by the robotic arm. It was observed that when the robotic arms R and T joints were close to collinear alignment one of the intermediate path points would occasionally cause the robot to switch to an edge case of the joints range to achieve the middle position. For example, a pose ideally solved with joints R and T in their zero-positions can also be met by a full revolution in opposing directions by both R and T.

4.4 – UnSolved Maze in Unknown Position

A maze was presented to the work cell and activated. No information regarding the pose of the maze or solution path was provided to the robot. The robotic system successfully identified the maze in space using the machine vision and self-commanded motion to place the robot pointer TCP at the start of the maze with a vertical offset. For visualization purposes, a marker was attached to the EEF and the maze replaced with a blank canvas. The resultant path was traced onto a sheet of paper as shown in Figure 26.

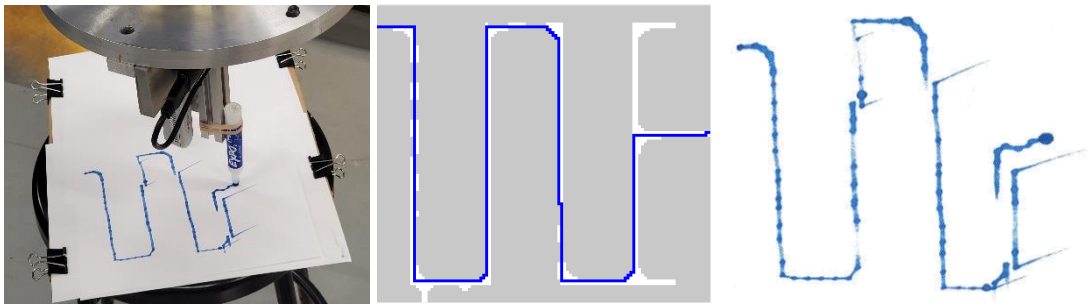

Figure 26: Automatically Identified and Solved Maze

Analyzing the rightmost image in Figure 26, a scan of the traced path by the robot, it is clearly identifiable that the robot briefly pauses at each path point. Further, issues where the end effector is rotated off the paper are shown where streaked lines extend off the path at various intervals. This may be root caused to the MoveIt motion planner which does not account for these issues. Subsequent testing showed that with tighter spacing between points, the MoveIt planner to failed more frequently. Comparing the traced path to the original maze presented the robot, the path is accurate and maintains precision of +/- 5 cm to the center line of the original grid maze.

Chapter 5. Conclusion and Future Work

In this work, a functional ROS Industrial motion planning and control system was assembled and deployed to physical hardware. The developed techniques successfully assembled a planar path, processed it into a trajectory, and executed the motion using the MoveIt motion planning framework. Future work will transition motion planning frameworks to a planner better optimized to semi-constrained cartesian motion such as Descartes or Tesseract.

The vision system, powered by an Intel RealSense D435i, was successfully used to open a vision pipeline for image capture and, separately, visualization of the environment in RViz,. Future work will shift all machine vision usage into the ROS framework (ie. the ROS wrapped library vs the Python wrapper) to allow the depth data to be incorporated into a persistent collision environment.

The DREAM.3D image filtering pipeline was successful in identifying locators from a plain environment, but struggled with complex environments involving objects of colors similar to the locator dots. As the locator dots and rotation were requirements derived from the actual maze path solver and not the path planner, future work will focus on developing a robust technique for defining a path in the real world and identifying that path in the machine vision for later navigation tasks.

The path planner developed for this study was constrained to planar surfaces. Future work will expand the capability of this system to curved surfaces. Exploration will consider the feasibility of projecting planar path plans onto curved surfaces.

Bibliography

[1] I. Malavolta, G. Lewis, B. Schmerl, P. Lago and D. Garlan, "How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering*, 2020.

[2] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software. 3*, 2009.

[3] K. M. Lynch and F. C. Park, Modern Robotics: Mechanics, Planning, and Control, Cambridge University Press, 2017.

[4] D. Coleman, I. A. Sucan, S. Chitta and N. Correl, "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study," *Journal of Software Engineering for Robotics,* vol. 5, no. 1, p. 3–16, May 2014.

[5] S. Edwards, d. Solomon, J. Nicho, C. Lewis, P. Hvass, J. Zoss and C. Flannigan, "REP: I0003 Title: Cartesian Path Planner Interface Pipeline," May 2014.

[6] D. Coleman, M. Moll and A. Zelenak, "Guide to Cartesian Planners in MoveIt," 7 January 2021. [Online]. Available: https://picknik.ai/cartesian%20planners/moveit/motion%20planning/2021/01/07/guide-to-cartesian-planners-in-moveit.html.

[7] J. D. Maeyer, B. Moyaers and E. Demeester, "Cartesian path planning for arc welding robots: Evaluation of the descartes algorithm," *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA),* pp. 1-8, 2017.

[8] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow and P. Abbeel, "Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization," *Robotics: Science and Systems,* 2013.

[9] Southwest Research Institute, "Noether," [Online]. Available: https://github.com/ros-industrial/noether.

[10] Institut Maupertuis, "Bezier," [Online]. Available: https://github.com/ros-industrial-consortium/bezier.

[11] Intel, "Intel RealSense Product Family D400 Series," Datasheet: 337029-010, Feb. 2021.

[12] A. Grunnet-Jepsen and D. Tong, "Depth Post-Processing for Intel® RealSense™ D400 Depth Cameras," Intel, 2020.

[13] Intel, "Realsense-ROS," [Online]. Available: https://github.com/IntelRealSense/realsense-ros.

[14] M. A. Groeber and M. A. Jackson, "DREAM.3D: A Digital Representation Environment for the Analysis of Microstructure in 3D.," *Integrating Materials 3*, pp. 56-72, 2014.

Appendix A. End Effector Design

A custom end effector was constructed for this project. To achieve both camera vision and a 'pointing' ability, the EEF includes two TCPs with unique reference frames. The end effector is mounted to the robot flange and defined relative to the Tool0 reference frame.

Two TCPs are defined:

- Camera TCP
- Pointer TCP

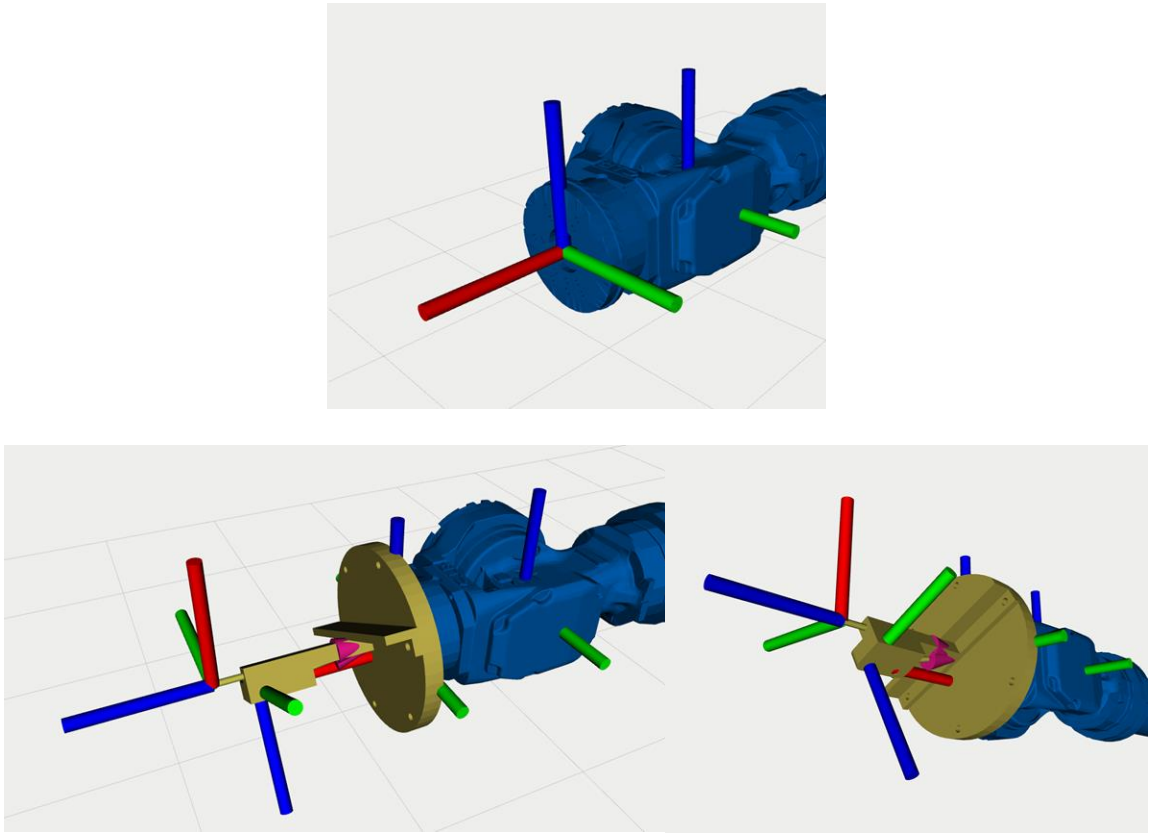

Figure 27: End Effector Reference Frames

Appendix B. Lab Specific Environment Setup Information

A Yaskawa Motoman MS210 industrial robot was used for physical hardware testing in this study. Prior to this study, no support package specific to this robot had been generated. Initial work in this study authored this package working with Yaskawa Motoman engineers and the ROS Industrial community. The resultant package was contributed and merged into the public Motoman repository.

The MS210 robot in the AIMS lab is part of robotic cell containing 3-industrial robots connected to a single master controller. This required a unique setup of the robot support package and subsequent MoveIt configuration as the joint trajectory messages for each robot must be merged into a single message block. Such a setup was achieved through empirical testing. Information on this setup is available upon request to the AIMS Lab. The URDF transform tree for this robotic cell is included in Figure 28. Only the Motoman MS210 was actually used in the motion code for this project.

Additional specifications for the Motoman MS210 robot were derived from Motoman's documentation available on their website: https://www.motoman.com/en-us.

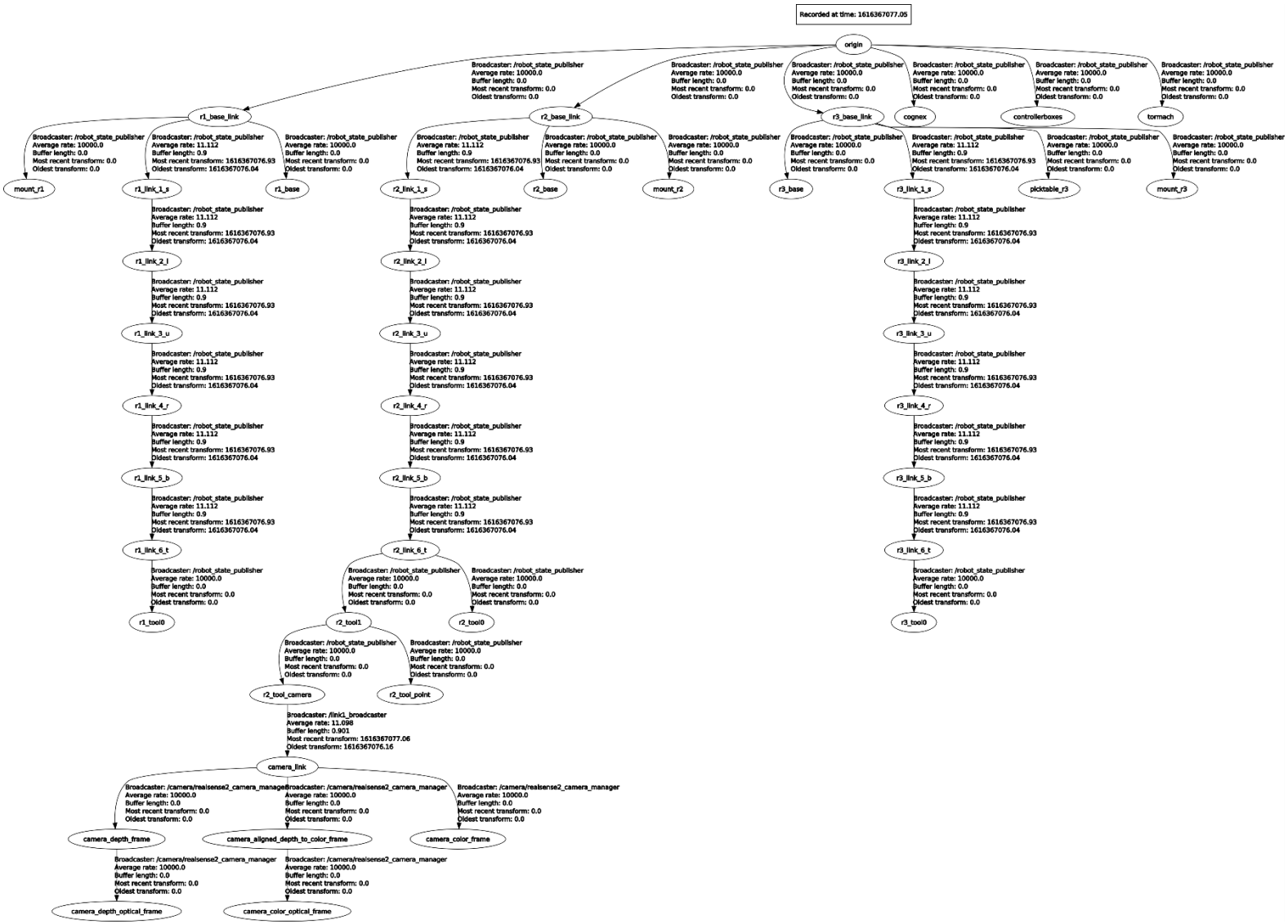

Figure 28 Complete Transformation Tree used by Industrial Robot System